UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1373

Programme 4
Robotique, Image et Vision

DATABASE ISSUES IN OBJECT-ORIENTED DESIGN

Gia Toan NGUYEN
Dominique RIEU

GROUPE DE RECHERCHE
GRENOBLE

Février 1991

RR-1373

DATABASE ISSUES IN OBJECT-ORIENTED DESIGN

NGUYEN Gia Toan[1] & Dominique RIEU[2]

[1] INRIA & [2] IMAG
Laboratoire de Génie Informatique
BP 53 X
38041 GRENOBLE Cedex
France

Tel : (33) 76.51.45.75

e-mail : nguyen@imag.fr

fax : (33) 76.44.66.75

ASPECTS BASES DE DONNEES EN CONCEPTION ORIENTEE OBJET

Résumé

On décrit un modèle de représentation de connaissances dédié aux applications de conception. L'objectif est de définir, réaliser et expérimenter un système basé sur une approche "objet" et capable de gérer la dynamique de ces applications. Le modèle s'inspire des techniques de l'intelligence artificielle, de la programmation par objets et des applications d'ingénierie assistée par ordinateur. Il met en oeuvre des concepts émanant des modèles à base de frames et des langages à objets. Sa définition tient compte également des besoins d'applications qui gèrent des objets évolutifs comme la CAO.
Ces besoins sont tout d'abord examinés. L'accent est mis sur les objets composites et les objets évolutifs. Les avantages et inconvénients respectifs des modèles actuels sont ensuite analysés. L'approche par objets est spécialement étudiée.
Le modèle SHOOD est enfin décrit. C'est un compromis entre les concepts d'objets et les techniques de représentation de connaissances. Son adéquation aux applications de conception est analysée. L'accent est mis sur les relations sémantiques, par exemple les dépendances entre objets, et sur l'évolution des objets.

Mots-clés : représentation de connaissances, programmation par objets, conception assistée.

DATABASE ISSUES IN OBJECT-ORIENTED DESIGN

NGUYEN Gia Toan[1] & Dominique RIEU[2]

[1] INRIA & [2] IMAG
Laboratoire de Génie Informatique
BP 53 X
38041 GRENOBLE Cedex
France

Tel : (33) 76.51.45.75

e-mail : nguyen@imag.fr

fax : (33) 76.44.66.75

Abstract

A knowledge representation model which is dedicated to design applications is described. The aim is to define, implement and experiment a system supporting the dynamics of design applications, based on the object paradigm. The model draws on artificial intelligence techniques, on object-oriented programming and engineering design applications. It includes concepts found in frame-based knowledge representations and object-oriented programming languages. It also takes into account the requirements of applications that support dynamically evolving objects e.g, CAD/CAM.
The requirements of engineering design applications are described first. The emphasis is on composite and evolving objects. The advantages and shortcomings of current modeling paradigms are analyzed. The object-oriented approach is specifically considered. The model SHOOD is finally described. It appears as a compromise between object-oriented concepts and knowledge representation techniques. Its ability to model design objects is detailed. The emphasis is on the support for semantic relationships, e.g dependencies between components, and for object evolution.

Key-words : knowledge representation, object-oriented programming, design applications.

1. INTRODUCTION

Engineering design applications provide an interesting challenge to the database and knowledge base communities because they involve the management of large, complex and evolving objects. This concerns various domains like VLSI circuits, architectural design and aircraft prototyping.
On the one hand, intricate semantic relationships involve usually numerous components which participate in the overall design. Structural complexity is therefore intimately mixed with semantic complexity. It is shown in the sequel that a confusion of both aspects exists in almost all knowledge representation and object-oriented languages, not to speak about previous data models. It is our opinion that design requirements require a clear distinction between the structural definitions and the semantic relationships.
On the other hand, the specific nature of the design process involves trial and error cycles which involve various alternatives. They may eventually appear untractable. The complexity of the design objects is therefore supported by a dynamic design process which produces partially complete and temporarily inconsistent artifacts.
The challenge is therefore twofold for the designers of knowledge base systems :
- they must implement a powerful modeling paradigm to cope with the evolving objects,

- they must simultaneously provide a flexible framework to support the design process, in order to manage :
- the multiple representations of the objects,
- the successive versions and alternatives of each particular object,
- the specific nature of the design artifacts i.e, their evolving structure, relationships and values whilst the design is in progress.

This complexity is probably the reason why existing database and knowledge base systems support only part of these requirements. Versions servers, simulation tools and ad-hoc database systems storing object libraries are commonly used [KAT86]. But the inadequacy of relational database systems to support effectively engineering design applications is now widely recognized [KIM90]. Also, the lack of persistent store for existing knowledge representation systems and languages has motivated experiments to connect them to databases. However, the usual discrepancy between the databases and the knowledge representation languages is cumbersome. Relational databases provide only flat or embedded relations which are inadequate to model semantic information and complex hierarchical part-component structures [SCH88]. Entity-relationship data models often require an underlying relational database management system to store the data [CAS90]. Finally, knowledge representation paradigms lack the storage capability and concurrency control of commercially available database systems [STE86].

New programming languages and database systems have appeared recently e.g, object-oriented languages [GOL83, STR86] and databases [BAN88, COP84]. While adequate for Computer Aided Software Engineering (CASE) from which they originate, they unfortunately lack important features for design applications e.g, semantic relationships [DIT88, KIM90]. This has given rise to a number of research efforts intended to extend their functionalities [CAR89, DES90, DIA90, MUR90].

Considering these advantages and shortcomings, this paper describes a model called SHOOD, which is an attempt to provide a sophisticated modeling paradigm for applications involving evolving and composite objects. It is assumed that the objects may be temporarily incomplete and inconsistent, and are linked by intricate semantic relationships.

The model elaborates on object-oriented programming languages and on knowledge representation languages found in Artificial Intelligence [STE86]. It tries to avoid many confusions currently associated with the basic object concepts in object-oriented and knowledge representation paradigms. It also takes into account the requirements of engineering design applications. They are described in Section 2, considering both composite objects (Section 2.1) and evolving objects (Section 2.2). The various advantages and shortcomings of existing data models are considered. The object-oriented paradigm is specifically considered in Section 3. The model provided by SHOOD is described in Section 4. It appears as a compromise between the knowledge representation and the object-oriented languages. Its ability to model design objects is detailed in Section 4.2. The emphasis is on the support for semantic relationships between objects and their dynamic evolution. Similarities and differences between engineering design and CASE applications are briefly mentioned where appropriate. Section 5 is a conclusion.

2. REPRESENTING DESIGN OBJECTS

Design objects can be characterized in terms of structural complexity and semantic complexity. Structural complexity is often amenable to part/sub-parts hierarchies that model composite objects [BLA87, KIM89]. Semantic complexity is however not amenable to semantic relationships only. The semantics of the objects is usually disseminated in ad-hoc design rules and constraints verification programs, because the modeling paradigms available today do not permit all the semantic information to be implemented in a single framework [RIE86]. The emphasis is here on the structural complexity i.e, on composite objects hierarchies (Section 2.1) and on the evolution of the design objects (Section 2.2).

2.1 Composite objects

Composite objects construction is the rationale for design applications [GIA90]. In contrast with business applications, the incremental nature of the design process involves the integration of existing parts or the design of new components that will ultimately form the required objects.

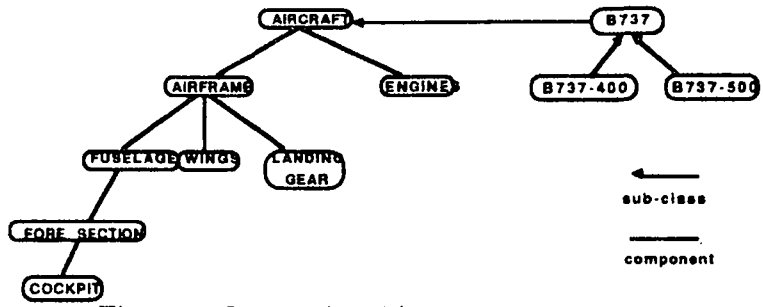

Figure 1. Composite objects.

New artifacts may involve only specific modifications to previous designs e.g, the B737 family includes the B737-300, B737-400 and the B737-500 models (Figure 1). The -100 and -200 models also exist but are no longer manufactured. They include more than 50% of similar equipment and components. The difference is in passenger capacity, engines, cockpit instrumentation and so on. They can generally be powered by different engine models e.g, CFM56 or V2500 for the Airbus A320. Further, marketing arguments make vendors put an emphasis on commonality between various models e.g, the B757 and the B767 have almost the same cockpit instruments, thus drastically reducing crew training and maintenance costs. Reuse and evolution of earlier designs is therefore a common practice.
Designs may also share subparts. For example, the components of an electronic flight management system (horizontal situation indicator, auto-pilot, fuel management system, ...) share the same redondant data bus. Sub-part definitions are therefore referenced by larger designs. This might put a real challenge on the designers because the available paradigms do not usually allow high-level component sharing. Further, instances of part/sub-part composition graphs are made of true hierarchies.
The ultimate goal is therefore to design a unified model that will support :

- part/sub-part agreggates i.e, composite objects,
- sub-part sharing,
- object reuse,
- object evolution,
- semantic relationships between objects and/or sub-parts (dependencies, etc...).

Existing modeling paradigms do not support simultaneously all these points, if at all.
Composite objects are difficult to model in existing database systems for many reasons. The CODASYL-like data models implement relationships through physical pointers which prohibit structure evolution and program modifications.
The relational model of data does not allow the definition of semantic links. It is "*not rich enough to admit the nested construction of complex designs or to properly capture the semantics of versions and representations of a design*" [KIM90]. Component hierarchies must be disseminated in multiple relations, thus confusing the implementation and the manipulation of the objects. All the semantics is contained in the application programs.
The entity-relationship model permits the definition of a limited form of semantic links : they may have specific attributes concerning the relationship roles and associated cardinality constraints. But their semantics has to be written in application programs. To some extent, the latter allow semantic information to be stored in the design objects : for example, an Airbus A320 has two engines. This can be stated in the cardinality constraints of a relationship between the aircraft and the engines. However, dependency relationships and value propagation among components is not allowed.
As described in Section 3, the object-oriented paradigm provides a more sophisticated notion of composite objects. However, "*the basic object-oriented concepts are not sufficient for*

modeling design objects" [KIM90]. For example, the semantics of composite objects is hard-wired and cannot be altered. There is a confusion between structural aspects - the sub-part hierarchies - and the semantic links that relate them - the dependencies between sub-parts. This severely limits the ability of these languages to perform design representation adequately.

2.2 Evolving objects

The design process is evolutionary by nature. Existing designs can be altered to produce new objects or to produce more satisfactory results. Database systems provide version and alternative mechanisms to support the multiple design instances [KAT86]. They put the burden on the user for handling the proper versions of the objects. Further, version derivation and merging for composite objects is the users' responsability.
Also, new customized components can be designed by trial and error cycles. This implies that they are not stabilized for long. That is, the definition of the design objects is not completed until some constraints and verification or simulation programs have been completed. Meanwhile, they can be inconsistent and only partially complete. Most database systems do not support adequately unknown and incomplete data. Systems providing integrity checking may in fact prohibit inconsistent data. This is in contradiction with the basic nature of the design process which requires a more sophisticated (and probably tunable) consistency control on the objects [BOR87]. In particular, constraint violations should be controlled by the user [KOT88].
In contrast, knowledge base systems often allow inconsistent data without user control, except when ad-hoc triggering mechanisms are implemented. They may even lack any control on the absence or inconsistency of the data : information may be untyped until it is actually created [STE86]. That is to say, they are too permissive and are in severe contradiction with usual database systems which are generally too restrictive concerning completeness and consistency issues. A compromise must therefore be found to allow a user-controlled form of data incompleteness and inconsistency in order to cope with the evolution of the design objects. A proposal is made in Section 4 concerning the model SHOOD.
Integrated CAD/CAM systems also handle design variants which may ultimately be chosen for the final design or which may be reworked for further testing and modifications. This is the price to pay for object reuse and evolution. For example, aircraft can be tested aerodynamically by numeric simulation programs, thus avoiding part of the long and costly wind-tunnel testing. The F-15 "Eagle" fighter wing was tested with multiple different designs. Similarly, the French "Rafale" fighter demonstrator was tested with 15 different wing configurations using extensive numeric simulations. The basic variants must therefore support alterations in their definitions i.e, in their structural and semantic definitions. This has also been recognized in CASE applications [DUC90].
Provision for these modifications is however rarely supported by existing database and knowledge base systems. CODASYL-like systems do not allow any evolution except for changing accordingly all the programs that use them. Relational models allow only addition of attributes to a relation, seldom deletion of attributes. Object-oriented data models allow modifications of the class definitions with the corresponding propagation on the object instances [PEN87, NGU89]. However, they provide only generic relationships between classes, thus prohibiting the modification of inter-object relationships. This is detailed in the next section.
Generally speaking, data evolution has been handled very crudly in database systems. Relational and object-oriented databases often provide version mechanisms. But they still place the burden on the user by offering complicated concepts without the corresponding assistance e.g, "generic" and "version" instances of the objects [KIM89].
Clearly, the dynamic evolution of the data requires more sophisticated mechanisms, alleviating the task of what the software engineering community call "configuration management" [DUC90]. This involves the definition, modification, documentation and retrieval of pieces of designs - all varying in time, and all related by semantic links and impacting on each other [AND90]. A first step towards the assistance to the user in managing object evolution in described in Section 4.2.

3. THE OBJECT-ORIENTED PARADIGM IN DESIGN

The object-oriented paradigm is often advertised as the most sophisticated and flexible approach for a large variety of applications, ranging from software engineering to office automation and database applications [GIA90]. Various features in object-oriented languages are discussed in this section to point out the potential advantages and shortcomings that are of interest for design applications. It appears that some basic functionalities are in contradiction with specific requirements of design applications, including semantic aspects and object evolution. Some advantages are detailed first.

3.1 Advantages

3.1.1 Abstraction

In the following, we shall distinguish between the notion of abstraction, which hides the implementation of the objects, and the notion of encapsulation found in the object-oriented paradigm. Encapsulation provides data abstraction but includes methods within object definitions to form their public interface [GOL83]. Abstraction is of interest in design application because it allows specific details to be hidden from the user. It allows also different variants to be used successively in composite objects without changing their definitions. For example, designers may be working on components which are not dependent on other sub-parts e.g the airframe and the engines of an aircraft. Abstraction is also a means to implement the notion of "grey boxes" to provide various levels of details to the designers [BLA87].

3.1.2 Reusability and sharing

Reusability is another important feature in object-oriented languages which is of great interest for design applications. This has been recognized in CASE applications also and the notion of "reusable element" is explicitly present in some advanced software engineering environments [DUC90]. Together with the notion of components in sub-part hierarchies, it provides the ability to abstract the objects at various levels of details. It also provides the opportunity to define clean interfaces and thus improve modularity and reusability.
For example, modern airliners may be fitted with integrated navigation equipment produced by different manufacturers. Inertial and navigation computers have standardized interfaces which may be connected to different displays in the so-called "glass cockpits" of today's airliners. This is also the reason why several engine models are usually available for an aircraft : control and monitoring equipment are somehow standardized. Clearly, abstraction also favors reuse of components since the redesign of existing parts is no longer necessary.

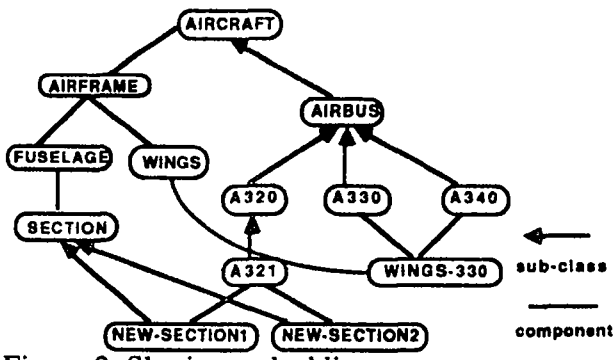

Figure 2. Sharing and adding components.

For example, the upcoming Airbus A330 and A340 share the same wing design. They may refer to the same class Wings-330 - a sub-class of Wings, itself a component of the class Airframe, which is in turn a component of the class Aircraft (Figure 2). Similarly, the A321 is a stretched version of the earlier A320 model fitted with two new fuselage sections fore and aft

of the wing (Figure 2). These two sections can be created as new sub-classes of the class Section - itself a component of the class Fuselage.

3.1.3 Specialization

Inheritance is also used in object-oriented models to implement a specialization relationship between objects. Objects can therefore be refined and incrementally defined using more specific constraints and new attributes. This can be straightforwardly adapted to design applications.
A systematic use of built-in inheritance imposes some constraints however, because it is never clear where the specialization process must stop. For example, each particular aircraft delivered is indeed a most specific specialization of some standard model. Within each specific airline carrier, dozens of A320 aircraft may only differ by their serial number. From a strict object-oriented perspective, they should be modeled by instances of a generic A320 model, implemented itself by an aircraft class. This is fundamentally a design decision which is not facilitated by the object-oriented paradigm. It is the result of the dichotomy existing between classes and their instances. It does not exist in paradigms based on prototypes [STE86]. This is one of its main drawbacks and probably the main challenge that object-oriented fanatics have to face. Some other problems follow.

3.2 Shortcomings

3.2.1 Encapsulation

A systematic use of the abstraction and encapsulation mechanisms, although adequate for a clean implementation, can be cumbersome but for one single reason : access to the values of object attributes would require the definition of many specific read and write operations. This is of course burdening, although it can be generated automatically for atomic data types (integer, strings, etc) [DEM87]. For example the age of an aircraft should be invoked by sending a message to the proper class, with the selector argument "age", the message argument the oid of the aircraft, and the receiver of the message the Aircraft class.
Calling methods to access attribute values is very time-consuming and can affect performance badly. Specific mechanisms have been proposed to extract attribute values elsewhere e.g, by a particular "*" prefix to attribute names [BAN88]. This syntactic sugar seems quite unatural. Record-based management systems do not require such artificial mechanism : invocation of the attribute's name is all what is needed. This is common sense, and do we really want such peculiarities in object-oriented systems ?
Another problem specifically related to encapsulation is the definition of methods within classes. Most programs in engineering design applications are external simulation, verification and display programs. Most of them already exist and need only be invoked where and when appropriate on the argument objects. Next, mixing code and object definition is in contradiction with design processes where much code operates on few object definitions. This point is of course irrelevant in CASE applications - where it all comes from - because the objects and the code are used together to produce large software environments [DUC90], whereas CAD/CAM applications only use the code to produce new design objects. In our opinion, this discrepancy is the reason for the first important mismatch between engineering applications and object-oriented concepts. Our conclusion is that programs or methods should be separated from object definitions.

3.2.2 Generic relationships

Another disadvantage of the object-oriented approach for modeling design objects is to support only generic relationships. They can only be defined between classes of objects, thus reducing the ability to implement specific relationships depending on the values of the instances being related. For example, dependency relationships can be defined in ORION between composite objects and their components [KIM89]. But the underlying semantics for this dependency is exclusive and existential : the deletion of the object implies the deletion of all its components. This is of course too strong for design applications where for example engine deletion on some

aircraft produces lots of spare parts that can be used to retrofit other engines. Instances should therefore be allowed to migrate from one class to another - say from the class Engine to the class SpareEngine - prior to the deletion of an aircraft. This in turn implies that the dependency between the aircraft and its engines be changed. Similarly, the semantics of composite objects in LOOPS implies that they can only be created if all their components are simultaneously created [STE86]. This is in contradiction with the design process where partially unknown objects must be worked out and later assembled together to form larger designs. The conclusion here is that dependency relationships should be decoupled from the part/sub-part graphs.

3.2.3 Multiple perspectives

Multiple perspectives or users points of view are seldom provided in object-oriented approaches [CAR89, MAR90]. They correspond to different ways by which the users perceive the objects : they correspond precisely to the notion of multiple representations found in knowledge representation languages [STE86]. They can be modeled by existing concepts in object-oriented languages e.g, specialization, aggregation and multiple-instanciation [NGU91].

4. A KNOWLEDGE REPRESENTATION APPROACH

The model SHOOD presented in this section is based on three principles :

P1 - every information is represented by an object,
P2 - every object is an instance of some other object,
P3 - any information concerning an object is stored as the value of an attribute.

The model is therefore reflexive and meta-circular i.e, it has access to its description and it is implemented using its own concepts [ESC90]. This approach provides the basis for extensibility. The model is implemented using two layers :

- the knowledge representation layer (KRL),
- the knowledge manipulation layer (KML).

A storage layer will also be provided to support object persistency. This will be achieved by interfacing with an existing object server (Figure 3).

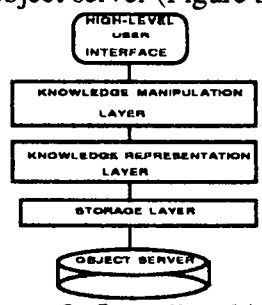

Figure 3. Overall architecture.

The KRL provides data abstraction, but does not provide encapsulation. Interface with existing programs should therefore be made at this level because object structures are made visible. The KML provides encapsulation and method invocation based on message passing techniques. The interface with design programs can therefore be implemented at both layers, providing the opportunity to access the data from both the usual attribute level of the objects, resembling the record-field access methods, and the encapsulated - method based - layer for sophiticated programming paradigms.

The rationale behind this approach is to provide the designer with the usual attribute-based representation of data which is in our opinion mandatory for CAD/CAM applications. It also provides higher level abstraction mechanisms in the KML, should the applications require casual user query facilities and graphics or other high level design facilities.

4.1 Concepts

In the following, the term object stands for object instance. An object instance belongs to one or more classes that define its structure and behavior.
The KRL includes the fundamental characteristics of frame-based knowledge representation languages, combined with concepts found originally in object-oriented languages. As such, the KRL merges functionalities usually not provided simultaneously in either paradigms.
Among them stand the usual notions of class, meta-class, instance, multiple inheritance and method (Section 4.1.1 and 4.1.2). Outstanding features include a disjunction relationship between classes and semantic relationships which are either dependency relationships between objects (Section 4.1.3.1) or attribute propagation among composite objects (Section 4.1.3.2).
Further, the systematic use of the principles P1 to P3 for the design of SHOOD has given rise to a powerful, reflexive and extensible model. Its implementation is based on a kernel of approximately 10 meta-classes which form the bootstrap of the model. The basic concepts are then implemented as instances of these meta-classes [ESC90].
The implementation is in Lisp. The extensive use of the kernel and of its meta-classes support a powerful representation language where the methods, attributes, inferences and constraints are themselves classes which, following the principle P2 above, are instances of specific meta-classes in the kernel. The invocation of a particular method or inference and the value of an attribute are instances of the classes which model methods, inferences and attributes. Besides its very condensed and easy to develop kernel, this provides the basis for extensions and evolution of the model itself (Section 4.3).
The actual concepts implemented by the model are generated from the bootstrap to provide the user with a basic set of functionalities e.g, set, list and bag manipulations together with the usual notions of class, inheritance, instance and composite objects. An emphasis is put on object evolution and its control by the designers (Section 4.2).

4.1.1 Attributes and classes

Objects are modeled in SHOOD by agregations of attributes e.g the length, wingspan, range and weight of an aircraft. They model either atomic values (integer, strings, etc) e.g range is 2,000 nautical miles (Figure 4), or complex components e.g the landing gear which includes a nose gear and two main gears (Figure 5). The scope of attributes is the class where they are first defined plus all their sub-classes. A sub-class may refine inherited attributes. It may not redefine them entirely, e.g give them a new domain class. Attribute refinement means for example more stringent constraints on the attribute. Attributes are defined by descriptors. There are three different descriptors attached to attribute definitions:
- type descriptors,
- constraint descriptors,
- inference descriptors.

Type descriptors define the domain of an attribute i.e the class where it takes on its values. Constraint descriptors define restrictions on the attribute values. Inference descriptors define the different ways an attribute value can be obtained. A partial order between the different inferences is required. Should an inference fail to provide the attribute value, the next inference in the list is activated. The conflict between different candidate inferences is solved by the method selection. This out of the scope of this paper. The user is always a default option i.e, if no other inference can provide the attribute value then the user is requested to give one.

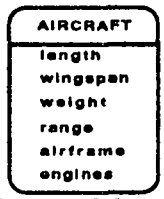

Figure 4. Attributes of the class Aircraft.

The complex components are themselves objects. This implies that components are referred to by identifiers. The identifiers, or OIDs, are unique within the system. Objects having the same attribute structure are grouped into classes e.g, the class Aircraft.

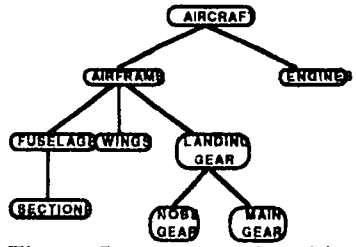

Figure 5. A composite object Aircraft.

4.1.2 Inheritance and disjunction

As provided by frame-based representation and object-oriented languages, the classes are structured in SHOOD along a super-class/sub-class relationship. It implements an inheritance relationship, by which sub-classes inherit all the attributes and methods of their super-classes. The semantics of inheritance in SHOOD is the strict set inclusion : a sub-class groups a set of instances which is a strict subset of the instances in the set represented by each of its super-classes.
SHOOD also features a disjunction relationship between classes. In contrast with the previous one, which is usually provided by object-oriented languages with various semantics, the disjunction relationship is genuinely provided in SHOOD to support generic concepts which are known to be distinct. This accelerate the classification mechanisms. It also inhibits name conflicts and several other features e.g, multiple inheritance and multiple instantiations. For example, aircraft and submarines are different and so far incompatible vehicles. They can therefore be defined by two sub-classes of the class Vehicle, namely Aircraft and Submarine which are related by a disjunction relationship (Figure 6). A side-effect of the disjunction relationship is that attributes defined in disjoint classes with the same name e.g "age", are considered different.

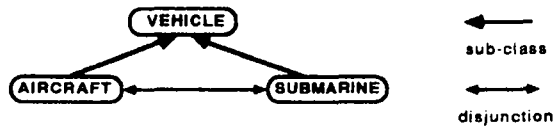

Figure 6. Inheritance and disjunction relationships.

A class may have several sub-classes which inherit its definition e.g, the class Aircraft has the sub-classes Long-Range, Short-Range which have the same attributes and which may refine them e.g the range attribute may only have specific values depending on the type of the aircraft. Subclasses may also add new attributes : Amphibian aircraft must have specific floating equipment (Figure 7).

Figure 7. Sub-class/super-class example.

A class may have several super-classes : multiple inheritance is supported. For example, the class Amphibian inherits the attributes of the classes Aircraft and Vessel. Because the attributes are inherited by all the sub-classes of a given class, name conflicts may appear in the attributes of a class having several super-classes. In SHOOD, such name conflicts are solved by an explicit inheritance of all conflicting attributes : they are prefixed by the name of the class which define them. For example in Figure 8, the class Amphibian inherits both the attribute "engine" of the class Aircraft and of the class Vessel. But it happens that naval engines have little to do with airborne engines, as well as the corresponding propellers. They are both

inherited and defined in the class Amphibian with their full name i.e their class name dot attribute name : "Aircraft.engine" and "Vessel.engine" respectively.
The semantics of inheritance is a strict set inclusion. This excludes exceptional instances i.e, objects which do not conform to the class definitions. This also implies that a class which has several super-classes models the intersection of the sets defined by the super-classes : an amphibian aircraft is both an aircraft and - a limited form of - a vessel. For example, the recent tilt-rotor aircraft "Osprey" V22 is both really an aircraft and a "rotor-craft"...
Due to the potential name conflicts involving inherited attributes, another facility in provided in SHOOD to avoid the duplication of attributes in a class when they represent the same concept. For example the classes Aircraft and Vessel depicted in Figure 14 have both an attribute "length" with the same semantics. The designer can define in a common super-class e.g Vehicle a so-called "deferred attribute". It defines a virtual attribute which cannot be instantiated in the class where it is defined, here Vehicle. It simply states that all the sub-classes from there down the inheritance graph will define this particular attribute "length" with the same semantics attached - up to disjunction of classes. The "length" attribute therefore need not be prefixed by any class name. It is implicitly here the Vehicle class where it appeared as deffered (Figure 8). It bears the same semantics wherever it appears in the sub-classes of Vehicle : the classes Aircraft, Vessel and Amphibian. In contrast, the attributes "engine" are different. They are inherited with their full name : "Aircraft.engine" and "Vessel.engine".

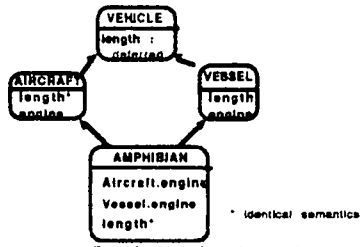

Figure 8. Full names for inherited attributes and deffered attributes.

4.1.3 Multiple instantiation

Multiple instanciation allows instances to belong simultaneously to different classes. This departs from the usual notion of instance in object-oriented approaches [GOL83]. It bears several advantages. First, it allows hollow classes to be avoided i.e classes with very few instances. This is a common problem in object-oriented modeling. Second it supports different simultaneous perspectives on the objects [NGU91]. Instances related to a particular class are seen by the user with the corresponding definition, and they can be seen simultaneously by other users as instances of other classes. We believe that it is easier to manage the "instance_of" relationship between object instances and their classes than to define new sub-classes modeling various points of view involving multiple inheritance, thus complicating the inheritance graph.

4.1.4 Semantic relationships

Semantic relationships are the most powerful and original feature in SHOOD. They elaborate on the relationships found in SRLwhere the user can define application dependent links, with their particular operations and attributes [FOX86]. Inheritance and disjunction are treated in SHOOD as particular cases of relationships. There is therefore no distinction between system defined and user-defined relationships in SHOOD. Two kinds of user-defined relationships are provided : dependency relationships and attribute propagation.

4.1.4.1 Dependency relationships

The dependency relationships are complementary to composite objects. They define the semantics of the part-component relationships which would otherwise be mere structural agregations of sub-parts. Recall that it is the case of all frame-based languages and that object-oriented languages impose an implicit and generic semantics to the composite obejcts. Semantic relationships are there mixed with the structural definitions of the composite objects. This

appears inadequate in many applications. As noted elsewhere [DIA90], various relationships must be available to the user. One of the most flexible system which offers user definable relationships is SRL. The goal in SHOOD is to provide a more limited although powerful set of links that the designers can use to model usual inter-object dependencies [ESC90]. There are four different kinds of such links : the exclusive, shared, existential and specific dependencies.

Exclusive dependency
The most simple and stringent form of dependency is the exclusive dependency. It allows a part to be the single owner of a component. In SHOOD, it allows an instance of the owner class to hold a privilege over an instance of the component class. The exclusive dependency can be defined between the corresponding classes i.e, in generic terms. It will then apply to all their instances. For example, an instance of the class Aircraft owns exclusively the instances of the class Engine which are fitted on its airframe. It should be noted however that no existential dependency is associated here. This is in contrast with most proposals in the literature [KIM89, STE86]. It is our opinion that the notion of existence should not be mixed with other notions of dependency as is usually the case : an aircraft may be equipped with a particular engine. The engine may be overhauled and later fitted on another aircraft. Or the previous owner aircraft may be scrapped and its engines reused on another. This is how it really happens. So there seem to be no reason to limit the flexibility of the dependencies by inappropriate confusion.
The notion of dependency is not necessarily linked to the notion of composite object either. A dependency may exist between two classes which are not linked by any component relationship. An aircraft belongs to a specific company, but it is not necessarily defined as a component of the company.

Shared dependency
The second form of dependency supported in SHOOD is the shared dependency. It is used to define a link between various parts and a particular object. Various airline carriers may for example share a particular hub in an airport. A particular side-effect is that the shared object cannot be deleted as long as any other object holds a dependency on it. This is a particular form of implicit existential dependency.

Existential dependency
The existential dependency is the most commonly used and provided in object-oriented languages. The main problem however is that it is always mixed with other kinds of relationships e.g composite objects. This confuses the design and the designers. In SHOOD, it can be used together with other relationships but it has to be explicitly defined by the user. As noted before, the existence of aircraft engines do not depend on the existence of aircraft, and vice-versa. But the existence of an aircraft is dependent on the existence of its airframe. Conversely, the airframe exists prior to the aircraft which is built around it. These differences have to be taken into account and are expressible in SHOOD. They were not in previous languages and representation models.

Specific dependency
A degenerated form of exception is supported concerning the dependencies. It is the specific dependency, which outrules the generic dependencies defined by classes. It states that between any particular instances of objects exist a dependency that does not follow the same pattern than the generic form. For example, a exclusive existential dependency may exist between the instances of helicopters - which can be otherwise defined as particular aircraft - and their rotor. These cannot be serviced and must be replaced periodically, as specified by the manufacturer. There is therefore a specific existential dependency between each particular rotor and the helicopter using it.

4.1.4.2 Attribute propagation

Attribute propagation is a feature allowing values from component attributes to be visible from various levels in the part hierarchy of composite objects. It is sometimes called horizontal or

selective inheritance [CAR89]. It is emphasized here that inheritance is a generic relationship in object-oriented languages, relating classes in the inheritance graph. It implements the super-class/sub-class relationship between classes. In contrast, the propagation of attribute values among objects is a instance specific relationships. It broadcasts attribute values among sub-parts of composite objects. It is therefore a relationships between instances, hence the "horizontal" qualifier, to differentiate from the "vertical" and generic inheritance relationship which is orthogonal.
Assuming that an airframe has an attribute "age", and that an aircraft is composed of an airframe and engines, the age of the aircraft may be inferred from its airframe's age. Due the fact that aircraft engines have to be overhauled periodically - typically every 2,000 hours - and that an airframe can be used 20 years or more, it is clear that selective propagation must be defined between components.
In the example, the aircraft age cannot be inferred from the engines' age. The designer must therefore propagate the airframe's "age" to the aircraft, excluding the "age" of the engines (Figure 9). Note that with attribute propagation, the class Aircraft need not have an attribute "age" defined. Again, this is an instance specific relationship, and the propagation must be done for each particular aircraft. From the designers point of view, the "age" of an aircraft will appear as if it had been defined for the class Aircraft.

Figure 9. Attribute propagation.

4.2 Modeling design objects

The modeling paradigm offered by SHOOD is threefold. It gives the ability to represent objects using three combined perspectives (Figure 10) :
- a semantic perspective, implementing the semantic relationships between object components,
- an structural perspective where objects can be altered, refined and reuse existing definitions, implementing the structural definitions of composite objects,
- a variant perspective where the evolution of the objects is modeled. It copes with the structural, semantic and content evolution of the objects using object versions.

While the inheritance perspective does not introduce new concepts with respect to existing object-oriented and frame-based languages, it is treated like other relationship in SHOOD. The semantic perspective includes the notions of dependency relationships and attribute propagation. The semantic perspective includes all semantic relationships because the composition relationship is decoupled from the notion of dependency between components. The variant perspective includes the notion of object evolution using the concepts of design improvement and degradation. It automatically generates versions of the successive designs produced.

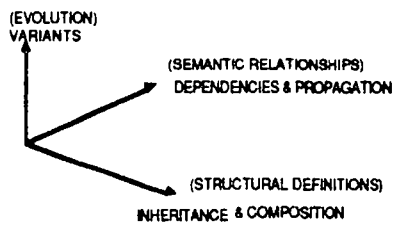

Figure 10. The three modeling perspectives in SHOOD.

Managing the various changes in the design objects, classifying them in the appropriate variants, finding the adequate variants to perform a specific design task and documenting them is called "configuration management" in CASE applications. While the model SHOOD does not provide extensive functionalities dedicated to this problem, some interesting features are available to handle the evolving nature of the designs. They are described in the following sections.

4.2.1 Evolving structures

Evolving structures may result from two main causes. The first one is the implementation of inappropriate class definitions. This is out of the scope of this paper. It is the "object-oriented design" problem. The second reason is reusability. It is of first importance here because it may result in the reuse of previous definitions for the design of new objects.
Evolving structures concern two different aspects. The first one concerns the operations available on the class definitions e.g adding an attribute. The second is the impact of these modifications on the corresponding instances. While the first aspect requires only appropriate code to implement the modifications of the class graph and is usually provided in some limited form by existing languages, the second requires carefull attention and specific mechanisms. It is usually very limited and unflexible.

Operations
The operations concerning class modifications can be classified in two categories : those modifying a single class e.g delete an attribute from a class, and those involving several classes e.g make a class a component or a subclass of another. Operations in SHOOD allow the modification of any part of an inheritance and composition graphs. Among the operations detailed elsewhere [NGU89] are the addition and deletion of super-class/sub-class and part/sub-part relationships.

Impact
Most important is the impact of the operations on the existing instances. A designer may want to leave the existing instances unchanged, and thus provoke the creation of a new version of the class. He may also wish to test the effect of the modifications performed on a class definition to all or part of the instances. A original feature is provided by SHOOD to support this form of evolution. It is based on the notion of relevant structure which corresponds to the potential combinations of attributes that can be generated from a class definition. Any modification performed on the original class definitions will be traced and the instances - whether incomplete and inconsistent - migrate and are attached to exactly one relevant structure. This allows the instances to be modified automatically according to the modifications performed on their class.

4.2.2 Evolving instances

Instance evolution is the most common aspect in data-intensive applications. In object-oriented paradigms, modification of attribute values may force instances to migrate. Available languages are usually very limited concerning this aspect because it requires a classification mechanism. In SHOOD, modification of the instances values triggers their propagation in the class graph.
The qualifying classes are all the super-classes of its current class, plus all their subclasses of which the modified instance meets the constraints. This is called instance propagation. It uses multiple instantiation, by which an instance may implicitly or explicitly belong simultaneously to several classes. This can be used as an assistance to the designer who may occasionnally wonder to which classes the object he is working on resembles most. For example previous regulations allowed transatlantic flights only to aircraft fitted with at least three engines, like the DC-10 or B747. Nowadays specific medium-range and twin engine aircraft like the B757 or Airbus A310 may be equipped for transatlantic flights under particular IATA regulations known as ETOPS ("Extended Twin-engine Operations"). It requires ad-hoc equipment e.g, Administration-specified engines. If modified accordingly, they will belong simultaneously to the Long-range and Medium-range classes.

4.2.3 Object improvement

The ultimate goal for all design process is to define objects that meet a given set of specifications, should this concern a space shuttle or a bicycle. The design evolves incrementally in stepwise manners but not linearly. Various alternatives may be explored in parallel. Older designs may also be deleted or reused and modified. Today, the control of this task is entirely left to the designers. They have to define with the appropriate tools and

expertise what defines good or better designs : greater speed, smaller space, lighter weight, etc.
An assistance is given by SHOOD concerning the evolution of the artifacts towards more complete and more consistent designs, with respect to the design specifications. Using the notion of relevant structure mentioned above (Section 4.2.1) the system generates automatically different versions of an object when its completeness or consistency improves. This is called an improvement cycle. Whenever the completeness and the consistency of the object does not improve, a degradation cycle is automatically created and hooked on the current improvement cycle. It can be deleted whenever the object reaches it back again after the appropriate modifications. For example, different wing configurations may be tested for an aircraft (Section 2.2). Those that improve the coefficient of lift CL and that decrease the coefficient of drag CD will be selected and stored as new versions in the improvement cycle. Those wing configurations that decrease CL or augment CD are attached to the degradation cycle (Figure 11).

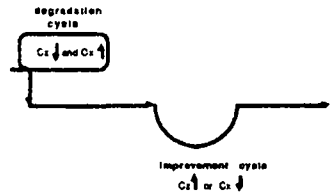

Figure 11. Improvement and degradation cycles.

4.3 Extensibility

An important aspect concerning the ability of the model to support evolution is extensibility. It provides for the extension of the concepts provided by the system or defined by the users. There is a facility in SHOOD which allows the designers to implement basic concepts by extending the system classes. This is done by the extensive use of the reflexive nature of the model. In particular, the basic concepts are all instances of metaclasses. They form the minimal structure of the system [COI87, KEE88]. They implement generic structures and behaviors for class and instance creation, modification and deletion. The users may adapt them by more specialized structures or behaviors. They can also extend this kernel to adapt it to the applications [PAT90]. This feature is fundamentally the support for the evolving nature of the design applications [ESC90].

5. CONCLUSION

The model SHOOD described in this paper appears to be a compromise between object-oriented modeling paradigms and knowledge representation languages found in Artificial Intelligence.
The rationale behind its design is the inadequacy of existing object-oriented languages to support flexible semantic relationships between objects as found in design applications. The ultimate goal is to implement a unified model to support :
- part/sub-part agregates i.e, composite objects,
- sub-part sharing,
- object reuse,
- object evolution,
- semantic relationships between parts or sub-parts (dependencies, attribute propagation, etc).

SHOOD is therefore an attempt to merge the advantages of both areas and to elaborate on existing techniques found in both paradigms. It includes features usually not provided simultaneously by existing knowledge representation and object-oriented languages e.g, meta-classes, disjunction, composite objects and powerful semantic relationships. Other outstanding features in SHOOD are :
- its ability to support evolving object structures,
- the management of incomplete and inconsistent objects,
- the ability to automatically support object evolution by the notion of object improvement,
- a uniform treatment of inheritance, disjunction, composition of objects and other semantic relationships e.g, dependencies.

The model therefore overcomes current limitations of object-oriented languages e.g, generic inheritance hierarchies or part-of relationships between object classes and built-in dependencies. It also overcomes limitations in existing knowledge representation languages by providing an automatic management of object evolution, concerning both the data structures, the completeness and the consistency of the design artifacts. Sophisticated method selection algorithms are being studied.
Experiments are underway in the project CIM-ONE for mechanical CAD/CAM, in cooperation with INSA, Ecole Centrale and Université Claude Bernard in Lyon (France).

Acknowledgements
The authors wish to thank Annie CULET, Chabane DJERABA and José ESCAMILLA for their contributions to the design and implementation of SHOOD. They also thank MM. B. DAVID, C. MARTY and D. VANDORPE for many invaluable discussions. This work is partially supported by INRIA and IMAG for the project SHERPA and by Région Rhône-Alpes for the project CIM-ONE.

REFERENCES

[AND90] ANDERSON B., GOSSAIN S. *Hierarchy evolution and the software lifecycle.* Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F). June 1990.
[BAN88] BANCILHON F. *Object-oriented database systems.* Proc. 7th ACM Symposium on principles of database systems. Austin (Texas). March 1988.
[BLA87] BLAKE E., COOK S. *On including part hierarchies in object-oriented languages, with an implementation in Smalltalk.* Proc. ECOOP '87. Paris (F). 1987.
[BOR87] BORNING A. & al. *Constraint hierarchies.* Proc. ECOOP '87. Paris (F). 1987.
[CAR89] CARRE B. *Méthodologie orientée objet pour la représentation des connaissances.* PhD Thesis. In French. Université de Lille Flandres-Artois (F). 1989.
[CAS90] CASANOVA M.A & al. *Algorithms for designing and maintaining optimized relational representations for entity-relationship schemas.* Proc. 9th Intl. Conf. on Entity-Relationship Approach. Lausanne (CH). October 1990.
[COI87] COINTE P. *Metaclasses are first class : the ObjVlisp model.* Proc. OOPSLA '87 Conf. Orlando (Fla). 1987.
[COP84] COPELAND G. & MAIER D. *Making Smalltalk a database system.* Proc. ACM SIGMOD Conf. Boston (Mass). 1984.
[DEM87] DeMICHIEL L., GABRIEL R.P *The Common Lisp Object System : an overview.* Proc. ECOOP '87. Paris (F). 1987.
[DES90] DESFRAY P. *A method for object-oriented programming : the class-relationship method.* Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F). June 1990.
[DIA90] DIAZ O. & GRAY P.M. *Semantic-rich user-defined relationship as a main constructor in object-oriented databases.* Proc. IFIP Conf. on Database Semantics "Object-Oriented Databases : Analysis, Design, Construction". North-Holland, 1990.
[DIT88] DITTRICH K. *Supporting semantic rules by generalized event/ trigger mechanisms.* Proc. Intl. Conf. Extending Database Technology. Venice (I). March 1988.
[DUC90] P. DUC. *ROSE/ADELE data models comparison.* Eureka Software Factory. February 1990.
[ESC90] ESCAMILLA J., JEAN P. *Relationships in an object knowledge representation model* Proc. 2nd Intl. Conf. Tools for Artificial Intelligence.Washington DC (USA). November 1990.
[FOX86] FOX M.S et al. *Experiences with SRL : an analysis of a frame-based knowledge representations.* Proc. 1st Int. Workshop on Expert Database Systems. Kiawah Island (South Carolina). October 1986.
[GIA90] GIACOMETTI F., CHANG T.C *Object-oriented design for modelling parts, assemblies and tolerances.* Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F). June 1990.
[GOL83] GOLDBERG A., ROBSON D.*Smalltalk 80 : the language and its implementation.* Addison-Wesley Publ. Co. 1983.
[KAT86] KATZ R. & al. *Version modeling concepts for computer-aided design databases.* Proc. ACM SIGMOD Conf. 1986.

[KEE88] KEENS S.E *Object-oriented programming in Common Lisp. A programmer's guide to CLOS*. Addison-Wesley. 1988.
[KEMPER 87] KEMPER A. & al. *An object-oriented database system for engineering applications*. Proc. ACM SIGMOD Conf. San Francisco (Ca). May 1987.
[KIM89] KIM W. & al. *Composite objects revisited*. Proc. ACM SIGMOD Conf. 1989.
[KIM90] KIM W. *Object-oriented databases : definition and research directions*. IEEE Trans. on Knowledge and Data Engineering. Vol. 2, n° 3. September 1990.
[KOT88] KOTZ H. *Supporting semantic rules by a generalized event/trigger mechanism*. Proc. Intl. Conf. Extending Database Technology. Venice (I). 1988.
[MAR90] MARINO O. *Multiple perspectives and classification mechanism in object-oriented representations*. Proc. ECAI Conf. Stockholm (S). July 1990.
[MUR90] MURPHY G.C & al. *Genericity, inheritance and relations : a practical perspective*. Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F). June 1990.
[NGU87] NGUYEN G.T, RIEU D. *Expert database concepts for engineering design*. Artificial Intelligence for engineering design, analysis and manufacturing. Vol. 1, n° 2. Academic Press.1987.
[NGU87] NGUYEN G.T, RIEU D. *Expert database support for consistent dynamic objects*. Proc. 13th Intl. Conf. on Very Large Data Bases. Brighton (G-B). September 1987.
[NGU89] NGUYEN G.T, RIEU D. *Schema evolution in object-oriented database systems*. Data & Knowledge Engineering, North-Holland. Vol. 4, n° 1. July 1989.
[PAT90] PATON N.W, DIAZ O. *Metaclasses in object-oriented databases*. Proc. IFIP Conf. on Database Semantics "Object-Oriented Databases : Analysis, Design, Construction". North-Holland, 1990.
[PEN87]PENNEY D.J & STEIN J. *Class modification in the GemStone object-oriented DBMS*. Proc. OOPSLA '87 Conf. Orlando (Fla). 1987.
[RIE86] RIEU D., NGUYEN G.T. *Semantics of CAD Objects for Generalized Databases*. Proc. 23rd Design Automation Conference, Las Vegas (USA). June 1986.
[RIE88] RIEU D., NGUYEN G.T *Dynamic schemas for engineering databases*. 4th Intl Conf. on Systems Research, Informatics and Cybernetics. Baden-Baden (FRG). August 1988.
[RIE91] RIEU D., NGUYEN G.T *Multiple instantiation for object representations*. Submitted for publication.
[SCH88] SCHEK H. *Nested relations, a step forward or backward?* IEEE Data Engineering. Vol. 11, n° 3. September 1988.
[STE86] STEFIK M., BOBROW D.G *Object-oriented programming: themes and variations*. The AI magazine. January 1986.
[STR86] STROUSTRUP B. *The C++ programming language*. Addison-Wesley. 1986.
[ZDO85] ZDONIK S.B *Object management systems for design environments*. IEEE Database Engineering. Vol. 8, n° 4. 1985.

Imprimé en France
par
.l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399